React Advance

Parent Child Communication

There are two directions a data can go and it's the following:

1. From parent to child
2. From child to parent

#Passing data from parent to Child

the parent component passes *props* to the child component and the child can then access the data from the parent via *this.props*.

Let's see an example

if you have two function components instead of class components, you don't even need to use props

Passing the data from the child to parent component is a bit trickier.
In order to do this, you need to do the following steps:

1. Create a callback function in the parent component. This callback function will get the data from the child component.

2. Pass the callback function in the parent as a prop to the child component.

3. The child component calls the parent callback function using props.

```
class MessageReceiver extends React.Component{
   constructor()
   {
      super()
       this.state={
         message:""
       }
  this.getMessage=this.getMessage.bind(this)
      }

    getMessage(msg)
    {
     this.setState({message:msg})
     console.log("get message")
    }


    render()
    {
      return <div>

           <h2>Parent Component</h2>
           <h3>{this.state.message}</h3>
            <MessageSender  messagefunc={this.getMessage}/>
     </div>
    }
```

```
}

export default MessageReceiver
```

```
import React from 'react'
class MessageSender extends React.Component{
   constructor()
   {
      super()
      this.sendMessage=this.sendMessage.bind(this)
   }

   sendMessage(str)
   {
 this.props.messagefunc(str)
   }


   render()
   {
      return <div>
          <h3>Child Component</h3>

            <button onClick={()=>{
              this.sendMessage("Hello from child")
            }}>Send Message to parent </button>
```

```
      </div>
    }
}

export default MessageSender
```

Best practices

1.Call back has to be configured as a parent function' reference to be passed to child function props

2.Any Event can be configured on child side

3.State management has to be taken care on parent side for data being received

4.This methodology is for simpler components ,for complex operations ,we can use react context

Higer order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.
Concretely, a higher-order component is a function that takes a component and returns a new component.

```
const EnhancedComponent =
higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's connect and Relay's createFragmentContainer.

React Ref

React refs allow us to access DOM properties directly. Normally, React uses state to update the data on the screen by re-rendering the component for us. But, there are certain situations where you need to deal with the DOM properties directly, and that's where refs come in clutch.

An example would be auto-focusing a text box when a component renders. React doesn't provide an easy way to do this, so we can use refs to access the DOM directly and focus the text box whenever the component renders on the screen

Error Boundaries

A JavaScript error in a part of the UI shouldn't break the whole app. To solve this problem for React users, React 16 introduces a new concept of an "error boundary".
Error boundaries are React components that **catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI** instead of the component tree that crashed. Error boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below them.

A class component becomes an error boundary if it defines either (or both) of the lifecycle methods `static`

`getDerivedStateFromError()` or `componentDidCatch()`. Use `static getDerivedStateFromError()` to render a fallback UI after an error has been thrown. Use `componentDidCatch()` to log error information.

Error boundaries work like a JavaScript `catch {}` block, but for components. Only class components can be error boundaries. In practice, most of the time you'll want to declare an error boundary component once and use it throughout your application.

Error boundaries do **not** catch errors for:

- Event handlers
- Asynchronous code
  (e.g. `setTimeout` or `requestAnimationFrame` callbacks)
- Server side rendering
- Errors thrown in the error boundary itself (rather than its children)

React Lazy Loading

It is a new function in react that lets you load react components lazily through code splitting without help from any additional libraries. Lazy loading is the technique of rendering only-needed or critical user interface items first, then quietly unrolling the non-critical items later. It is now fully integrated into core react library itself. We formerly used *react-loadable* to achieve this but now we have *react.lazy()* in react core.

Suspense

Suspense is a component required by the lazy function basically used to wrap lazy components. Multiple lazy components can be wrapped with the suspense component. It takes a fallback property that accepts the react elements you want to render as the lazy component is being loaded.

Why is Lazy Loading (& Suspense) Important

Firstly, bundling involves aligning our code components in progression and putting them in one javascript chunk that it passes to the browser; but as our application grows, we notice that bundle gets very cumbersome in size. This can quickly make using your application very hard and especially slow. With Code splitting, the bundle can be split to smaller chunks where the most important chunk can be loaded first and then every other secondary one lazily loaded.

Also, while building applications we know that as a best practise consideration should be made for users using mobile internet data and others with really slow internet connections. We the developers should always be able to control the user experience even during a suspense period when resources are being loaded to the DOM.